Department of
Information Engineering
and Computer Science
DISI

UNIVERSITY
OF TRENTO - Italy

DISI - Via Sommarive 14 - 38123 Povo - Trento (Italy)
http://www.disi.unitn.it

SIMULATION MATCHING IMPLEMENTATION OF AUTOMATA MODULO THEORY (AMT)

Fabio Massacci and Ida Siahaan

August 2009

Technical Report # DISI-09-074

Contents

1 Introduction **3**
1.1 The Contributions of the Paper 4

2 Security by Contract in a nutshell **4**

3 Automata Modulo Theory **7**

4 Simulation **10**

5 Simulation Matching **14**

6 The Architecture **16**
6.1 Updating the S3MS ConSpec Parser 18
6.2 Extending the S3MS PolicyManager 19
6.3 Extending the S3MS Framework 19
6.4 Test Cases 19

7 Design Decisions **20**

8 Experiments on Desktop **21**

9 Related Works and Conclusions **23**

A Simulation Matching Prototype Class Diagram **26**

B Simulation Matching Prototype Experiments **27**

Abstract

The traditional realm of formal methods is the off-line verification of formal properties of hardware and software. In this technical report we describe a different approach using fair simulation for matching and adapts the Jurdziński's algorithm on parity games. The simulation algorithm takes as input two automata representing respectively the formal specification of a contract and of a policy. A match is obtained when every security-relevant action invoked by contract can also be invoked by Aut^P. In other words, every behavior of Aut^C is also a behavior of Aut^P.

In this paper we thoroughly describe contract-policy matching using simulation, a prototype made on .NET for Desktop PC and give some experimental results.

Keywords Formal Specification · Mobile Code · Language-based security · Malicious code · Security and privacy policies

Table 1: End Users' Distilled Security Requirements

USE of Costly functionalities Any invocation of paid services, such as sending text messages, using GPRS or wireless connections, must be controllable by the user.

NETwork connectivity Any external connections made by the application can be controlled.

PRIvate information management It is necessary to control what data is accessed by the application such as local files, PIM items or contacts from Contact List.

INTeraction with other applets This requirement makes necessary to control means of interprocess communication, in particular sockets and memory-mapped files.

Power consumption This requirement is two-fold: it makes necessary to control the invocation of power-consuming functionality, such as WiFi connections, and to control the battery level in course of running the application. This can be mapped into the NET and USE categories.

EXTended functionality If the device is equipped with some advanced functionality, such as camera or GPS receiver, its use is likely to be controlled by policies.

1 Introduction

In this technical report we describe a prototype implementation for matching the claims on the security behavior of a midlet (for short *contract*) with the desired security behavior of a platform (for short *policy*) for realistic security scenarios (such as the "only https connections").

The formal model considered for capturing contracts and policies is based on the novel concept of *Automata Modulo Theory* ($\mathcal{AMT}$). $\mathcal{AMT}$ has been introduced in [22], which extends Büchi Automata (BA) by labeling transitions with expressions belong to decidable theories. It is suitable for formalizing systems with finitely many states but infinitely many transitions by leveraging on the power of satisfiability-modulo-theory (SMT) decision procedures. In this way we can represent the task of matching the contract with the policy as language containment problem between two automata. However, while [23] provides the theoretical framework, namely simulation matching algorithm and the complexity results of the operation, the actual implementation of the algorithm and the integration with a state-of-the-art theory solver is still left open.

Contracts and policies may vary significantly but a number of analyses of security requirements for mobile and ubiquitous applications [20, 29, 33] have shown that we can essentially distill them in few categories (Table 1). Such requirements can then be mapped into concrete behavioral constraints on usage of APIs. Here we discuss informally the syntax and refer to [1] for details.

The security behaviors provided by the contract and desired by the policy can be represented as automata where transitions correspond to invocation of security-relevant actions as suggested by Erlingsson [12, p.59] and Sekar et al. [27]. Then the operation of matching the midlet's claim with platform policy can be mapped into classical problems in automata theory.

One possible alternative is *language inclusion*: given two automata Aut^C and Aut^P representing respectively the formal specification of a contract and of a policy, we have a match when the execution traces of the midlet described by Aut^C is a subset of the

acceptable traces for Aut^P. To check this property we can complement the automaton of the policy, thus obtaining the set of traces disallowed by the policy and check its intersection with the traces of the contract. If the intersection is not empty, any behavior in it corresponds to a security violation, pursued in [22].

The other alternative is the notion of *simulation*: we have a match when every APIs invoked by Aut^C can also be invoked by Aut^P. In other words, every behavior of Aut^C is also behavior of Aut^P. Simulation is usually a stronger notion than language inclusion as it requires that the policy allows the actions of the midlet's contract in a "step-by-step" fashion, whereas language inclusion looks at an execution trace as a whole, pursued in [23].

In this technical report we use the approach of simulation as in [23], namely given two automata Aut^C and Aut^P representing respectively the formal specification of a contract and of a policy we have a match when there is no behavior of Aut^C which is disallowed by Aut^P.

1.1 The Contributions of the Paper

We discuss the overall implementation architecture and the integration issues with a state of the art decision procedure solver NuSMV [9] integrated with its MathSAT libraries [7]. This report is based on simulation algorithm implementation project report [31] and experiment from [6]. The implementation extended the S3MS Framework [10] by implementing the Simulation Matching Algorithm (SMA) [23].

To this extent we have decided to implement simulation as game graph with oracle calls to the decision procedures available in NuSMV. Therefore our design decision $\mathcal{AMT}$ makes reasoning about infinite transitions systems with finite states possible without symbolic manipulation procedures of zones and regions or finite representation by equivalence classes whose memory intensive characteristic is not suitable for our application.

The second contribution is a performance analysis of the integration design alternatives regarding the construction of expressions, the initialization of solver, and the caching of temporary results by considering both running time and internal metrics of various available options.

First, we introduce the concept of Automata Modulo theory (Section3). We continue by briefly recapping the notion of $\mathcal{AMT}$ simulation (Section4). After description of the simulation algorithm for contract policy matching we introduce the architecture of our prototype (Section6) and the design decisions needed for evaluation (Section7). Finally we report our experimental findings (Section8) and conclude with a brief discussion of related work (Section9).

2 Security by Contract in a nutshell

Security-by-contract (S×C)[11, 5] proposed to augment mobile code with a claim on its security behavior that can be matched against a mobile platform policy on-the-fly, which provides semantics for digital signatures on mobile code. In an S×Cframework [11, 5] a mobile code is augmented with a claim on its security behavior (an *application's contract*) that could be matched against a mobile *platform's policy* before downloading.

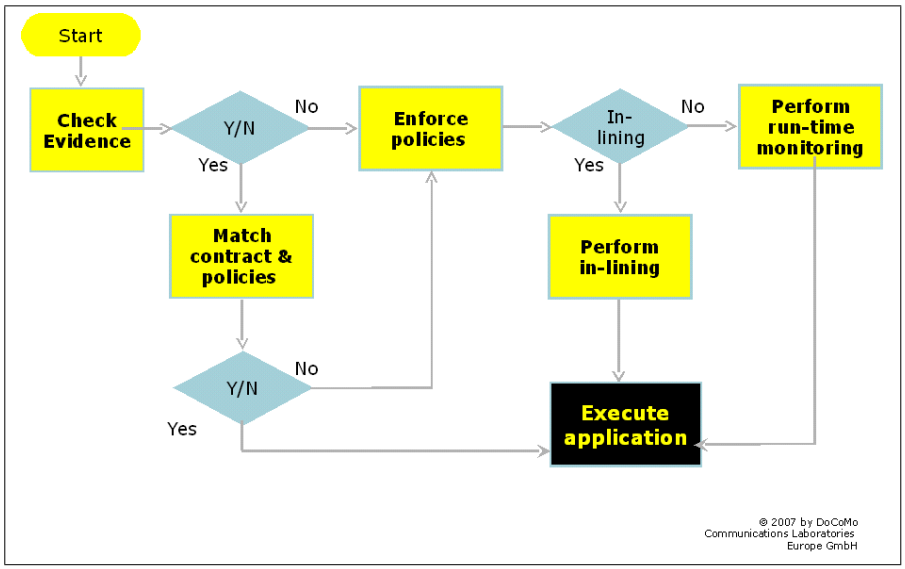


Figure 1: Workflow in Security-by-Contract

At *development time* the mobile code developers are responsible for providing a description of the security behavior that their code finally provides. Such a code may undergo a formal certification process by the developer's own company, the smart card provider, a mobile phone operator, or any other third party for which the application has been developed. By using suitable techniques such as static analysis, monitor in-lining, or general theorem proving, the code is certified to comply with the developer's contract. Next, the code and the security claims are sealed together with the evidence for compliance (either a digital signature or a proof) and shipped as shown on Figure 2.

At *deployment time*, the target platform follows a workflow as depicted in Figure 1 [5]. This workflow is a modification of S×Cworkflow [5]) by adding optimization step. First, the correctness of the evidence of a code is checked. Such evidence can be a trusted signature [32] or a proof that the code satisfies the contract (one can use Proof-Carrying-Code (PCC) techniques to check it [24]). When there is evidence that a contract is trustworthy, a platform checks, that the claimed contract is compliant with the policy to enforce. If it is, then the application can be run without further ado. It is a significant saving from in-lining a security monitor. In case that at *run-time* we decide to still monitor the application, then we add a number of checks into the application so that any undesired behavior can be immediately stopped or corrected.

Matching succeeds, if and only if, by executing an application on the platform, every behavior of the application that satisfies its contract also satisfies the platform's policy. If matching fails, but we still want to run the application, then we use either a security monitor in-lining, or run-time enforcement of the policy (by running the application in parallel with a reference monitor that intercepts all security relevant actions). However with a constrained device, where CPU cycles means also battery consumption, we need to minimize the run-time overheads as much as possible.

A *contract* is a formal specification of the behavior of an application for relevant

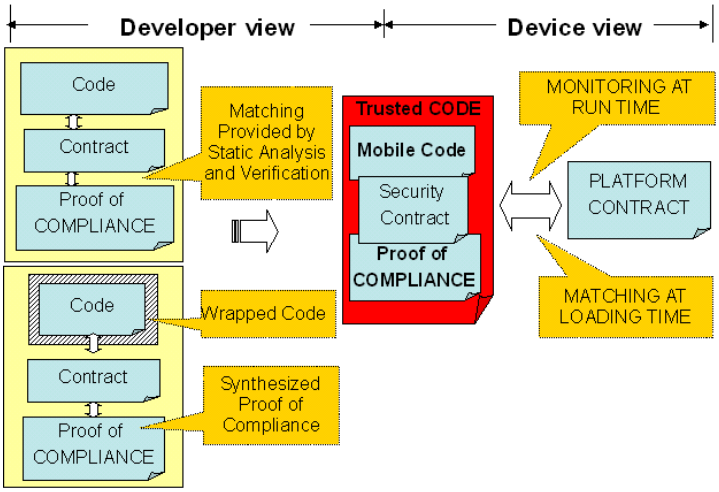


Figure 2: Mobile Code Components with Security-by-Contract

security actions for example Virtual Machine API Calls, Web Messages. By signing the code the developer certifies that the code complies with the stated claims on its security-relevant behavior. A *policy* is a formal specification of the acceptable behavior of applications to be executed on a platform for what concerns relevant security actions. Thus, a digital signature does not just certify the origin of the code but also bind together the code with a contract with the main goal to provide a semantics for digital signatures on mobile code. Therefore, this framework is a step in the transition from trusted code to trustworthy code.

Technically, a contract is a security automaton in the sense of Schneider [16], and it specifies an upper bound on the security-relevant behavior of the application: the sequences of security-relevant events that an application can generate are all in the language accepted by the security automaton.

A *policy*(also *contract*) covers a number of issues such as file access, network connectivity, access to critical resources, or secure storage. A single contract can be seen as a list of disjoint claims (for instance rules for connections). An example of a rule for sessions regarding A Personal Information Management (PIM) and connections is shown in Example 2.1, which can be one of the rules of a contract. Another example is a rule for method invocation of a Java object as shown in Example 2.2. This example can be one of the rules of a policy. Both examples describe safety properties, which are common properties to be verified.

Example 2.1 *PIM system on a phone has the ability to manage appointment books, contact directories, etc., in electronic form. A privacy conscious user may restrict network connectivity by stating a policy rule: "After PIM is opened no connections are allowed". This contract permits executing the* `javax.microedition.io.Connector.open()` *method only if the* `javax.microedition.pim.PIM.openPIMList()` *method was never called before.*

Example 2.2 *The policy of an operator may only require that "After PIM was accessed only secure connections can be opened". This policy permits executing the* `javax.microedition.io.Connector.open(string url)` *method only if the started connection is a secure one i.e.* `url` *starts with "https://".*

We can have a slightly more sophisticated approach using Büchi automata [28] if we also want to cover liveness properties as shown in the following Example 2.3.

Example 2.3 *If the application should use all the permissions it requests then for each permission p at least one reachable invocation of a method permitted by p must exist in the code. For example if p is* `io.Connector.http` *then a call to method* `Connector.open()` *must exist in the code and the url argument must start with "http". If p is* `io.Connector.https` *then a call to method* `Connector.open()` *must exist in the code and the url argument must start with "https" and so on for other constraints e.g. permission for sending SMS.*

3 Automata Modulo Theory

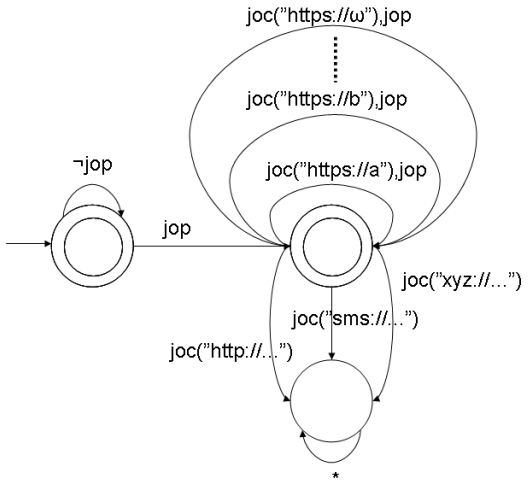
The security behaviors, provided by the contract and desired by the policy, can be represented as automata, where transitions corresponds to invocation of APIs as suggested by Erlingsson [12, p.59] and Sekar et al. [27]. Thus, the operation of matching the midlet's claim with platform policy can be mapped into classical problems in automata theory.

One possible mechanism to represent matching is *language inclusion*: given two automata Aut^C and Aut^P representing respectively the formal specification of a contract and of a policy, we have a match when the execution traces of the midlet described by Aut^C are a subset of the acceptable traces for Aut^P. To check this property we can complement the automaton of the policy, thus obtaining the set of traces disallowed by the policy and check its intersection with the traces of the contract. If the intersection is not empty, any behavior in it corresponds to a security violation.

The other alternative is the notion of *simulation*: we have a match when every APIs invoked by Aut^C can also be invoked by Aut^P. In other words, every behavior of Aut^C is also a behavior of Aut^P. Simulation is a stronger notion than language inclusion as it requires that the policy allows the actions of the midlet's contract in a "step-by-step" fashion, whereas language inclusion looks at an execution trace as a whole. We pursue the language inclusion approach in [22] and in this technical report and refer to [23] for the simulation approach.

While this idea of representing the security-digest as an automaton is almost a decade old [27, 12], the practical realization has been hindered by a significant technical hurdle: we cannot use the naive encoding into automata for practical policies. Even the basic policies in Ex. 2.1 and Ex. 2.2 lead to automata with infinitely many transitions.

Fig.3a represents an automaton for Ex. 2.2. We start from state p_0 and stay in this state while PIM is not accessed (jop). As PIM is accessed, we move to state p_1 and stay in state p_1 only if the started connection javax.microedition.io.Connector.open(string url) method is a secure one (url starts with "https://") or we keep accessing PIM (jop). If we start an insecure connection javax.microedition.io.Connector.open(string url), for example url starts with "http://" or "sms://", then we enter state e_p.



(a) An Infinite Automaton of Ex. 2.2

$$
\begin{aligned}
joc(url) &\doteq \texttt{javax.microedition.io.}\\
&\quad\ \texttt{Connector.open}(url)\\
jop &\doteq \texttt{javax.microedition.pim.}\\
&\quad\ \texttt{PIM.openPIMList}(\ldots)\\
p(url) = type &\doteq url.\texttt{startsWith}(type)
\end{aligned}
$$

(b) Abbreviations for Java APIs

Figure 3: Infinite Transitions Security Policies

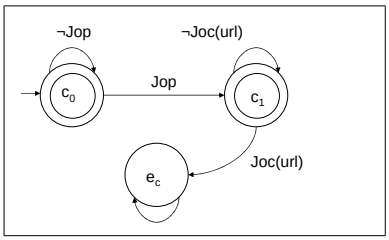
The examples presented are from a Java VM; since we do not consider it useful to invent our own names for API calls, we use the `javax.microedition` APIs (even though verbose) for the notation shown in Fig.3b.

Definition 3.1 (Automaton Modulo Theory ($\mathcal{AMT}$)) *An $\mathcal{AMT}$ is a tuple $A = \langle E, \mathcal{T}, \Sigma, S, \mathbf{s_0}, \Delta, F\rangle$, where E is a finite set of Σ-formulas in Σ-theory $\mathcal{T}$, S is a finite set of states, $\mathbf{s_0} \in S$ is the initial state, $\Delta \subseteq S \times E \times S$ is a labeled transition relation, and $F \subseteq S$ is a set of accepting states.*

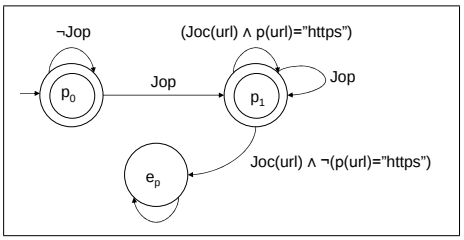
Figure 4 shows two examples of $\mathcal{AMT}$ using the signature for $\mathcal{EUF}$ with a function symbol $p()$ representing the protocol type used for the opening of a *url*. As described in the cited examples the first automaton forbids the opening of plain http-connections as soon as the PIM is invoked while the second just restricts connections to be only https.

The transitions in these automata describe with an expression a potentially infinite set of transitions: the opening of all possible *urls* starting with https. The automaton modulo theory is therefore an abstraction for a concrete (but infinite) automaton. The *concrete automaton* corresponds to the behavior of the actual system in terms of API calls, value of resources and the likes.

From a formal perspective, the concrete model of an automaton modulo theory intuitively corresponds to the automaton where each symbolic transition labeled with an expression is replaced by the set of transitions corresponding to all satisfiable instantiations of the expression. To characterize how an automaton captures the behavior of programs we need to define the notion of a trace. So, we start with the notion of a symbolic run which corresponds to the traditional notion of run in automata.



(a) $\mathcal{AMT}$ rule from Example 2.1



(b) $\mathcal{AMT}$ rule from Example 2.2

$$
\begin{aligned}
Joc(url) &\doteq \texttt{Joc(joc,url)} \\
Jop &\doteq \texttt{Jop(jop,}x_1,\ldots,x_n\texttt{)} \\
p(url) = type &\doteq url.\texttt{startsWith(}type\texttt{)} \\
joc &\doteq \texttt{javax.microedition.io.Connector.open} \\
jop &\doteq \texttt{javax.microedition.pim.PIM.openPIMList}
\end{aligned}
$$

`Joc,Jop` are predicate symbols representing respectively `joc(url),jop(`$x_1,\ldots,x_n$`)` APIs.

(c) Abbreviations for expressions

Figure 4: $\mathcal{AMT}$ Examples

Definition 3.2 ($\mathcal{AMT}$ symbolic run) *Let $A = \langle E, \mathcal{T}, \Sigma, S, \mathbf{s_0}, \Delta, F\rangle$ be an $\mathcal{AMT}$. A* symbolic run *of A is a sequence of states alternating with expressions $\sigma = \langle s_0 e_1 s_1 e_2 s_2 \ldots\rangle$, such that:*

1. $s_0 = \mathbf{s_0}$
2. $(s_i, e_{i+1}, s_{i+1}) \in \Delta$ *and* e_{i+1} *is $\mathcal{T}$-satisfiable, that is there is some Σ-structure $\mathcal{M}$ a model of Σ-theory $\mathcal{T}$ and there exists some assignment α such that $(\mathcal{M}, \alpha) \models e_{i+1}$.*

A finite symbolic run *is denoted by $\langle s_0 e_1 s_1 e_2 s_2 \ldots s_{n-1} e_n s_n\rangle$. An* infinite symbolic run *is denoted by $\langle s_0 e_1 s_1 e_2 s_2 \ldots\rangle$. A finite run is* accepting *if the last state goes through some accepting state, that is $s_n \in F$. An infinite run is* accepting *if the automaton goes through some accepting states infinitely often.*

In order to capture the actual system invocations we introduce another type of run called *concrete run* which is defined over valuations that represent actual system traces. A valuation ν consists of interpretations and assignments which are actual system traces.

Definition 3.3 ($\mathcal{AMT}$ concrete run) *Let $A = \langle E, \mathcal{T}, \Sigma, S, \mathbf{s_0}, \Delta, F\rangle$ be an $\mathcal{AMT}$. A* concrete run *of A is a sequence of states alternating with a valuation $\sigma_C = \langle s_0 \nu_1 s_1 \nu_2 s_2 \ldots\rangle$, such that:*

1. $s_0 = \mathbf{s_0}$
2. *there exists expressions $e_{i+1} \in E$ such that $(s_i, e_{i+1}, s_{i+1}) \in \Delta$ and there is some Σ-structure $\mathcal{M}$ a model of Σ-theory $\mathcal{T}$ such that $(\mathcal{M}, \alpha_{i+1}) \models e_{i+1}$, where ν_{i+1} represents α_{i+1} and $\mathcal{I}(e_{i+1})$.*

A finite concrete run *is denoted by* $\langle s_0\nu_1 s_1\nu_2 s_2 \ldots s_{n-1}\nu_n s_n\rangle$. *An* infinite concrete run *is denoted by* $\langle s_0\nu_1 s_1\nu_2 s_2 \ldots\rangle$. *A finite run is* accepting *if the last state goes through some accepting state, that is* $s_n \in F$. *An infinite run is* accepting *if the automaton goes through some accepting states infinitely often. The trace associated with* $\sigma_C = \langle s_0\nu_1 s_1\nu_2 s_2 \ldots\rangle$ *is the sequence of valuations in the run. Thus a trace is* accepting *when the corresponding run is accepting.*

We use definition of run as in [14] which is slightly different from the one we use in [22], where we use only states.

Example 3.1 *An example of an* accepting symbolic run *of* $\mathcal{AMT}$ *rule from Example 2.2 shown in Figure 4b is*

c_0 $Jop(jop,file,permission)$ c_1 $Joc(joc,url)\wedge p(url)$="https" c_1 $Jop(jop,file,permission)$ c_1 $Joc(joc,url)\wedge p(url)$="https" ...

that corresponds with a non empty set of accepting concrete runs *for example*

c_0(*jop*,$PIM.CONTACT\_LIST$,$PIM.READ\_WRITE$) c_1 (*joc*,"https://www.esse3.unitn.it/")

c_1(*jop*,$PIM.CONTACT\_LIST$,$PIM.READ\_ONLY$) c_1 (*joc*,"https://online.unicreditbanca.it/login.htm") ...

Remark 3.1 *A symbolic run defined in Definition 3.2 is interpreted by a non empty set of concrete runs in Definition 3.3. This is a nature of our application domain where security policies define* $\mathcal{AMT}$ *in symbolic level and the system to be enforced has concrete runs. In other domains where we need the converse, namely to define symbolic runs from concrete runs, then a symbolic run defined in Definition 3.2 can be considered as an abstraction of concrete runs by Definition 3.3.*

4 Simulation

The notion of *simulation* in $\mathcal{AMT}$ is both *fair* and *symbolic*. The fairness in $\mathcal{AMT}$ is similar to *fair simulation* in Büchi automata as in [19]. A system fairly simulates another system if and only if in the simulation game, there is a strategy that matches each fair computation of the simulated system with a fair computation of the simulating system. Efficient algorithms for computing a variety of simulation relations on the state space of a Büchi automaton were proposed in [14] using parity game framework, that is based on small progress measures [21]. Another algorithm based on the notion of fair simulation was presented in [15]. The *symbolism* in $\mathcal{AMT}$ is similar to the theory of symbolic bi-simulation for the π-calculus [18]. This symbolic representation can express the operational semantics of many value-passing processes in terms of finite symbolic transition graphs despite the infinite underlying labeled transitions graph.

In the sequel we will use s to denote states of the application's contract and t to denote state of the platform's policy.

Definition 4.1 (Concrete Fair Compliance Game) *Let* A^c *and* A^p *be* $\mathcal{AMT}$ *with initial states* s_0 *and* t_0 *respectively. A* Concrete Fair Compliance Game $G^C_{A^c,A^p}(s_0,t_0)$ *is played by two players,* `Contract` *and* `Policy`*, in rounds.*

1. *In the first round* `Contract` *is on the initial state* $s_0 \in S^c$ *and* `Policy` *is on the initial state* $t_0 \in S^p$.

2. `Contract` *chooses a transition* $\langle s_i, e_i^c, s_{i+1}\rangle \in \Delta_{\mathcal{I}}^c$ *with a valuation* ν_i *represents* α_i *and* $\mathcal{I}(e_i)$ *such that* $(\mathcal{M}, \alpha_i) \models e_i^c$ *and moves to* s_{i+1}.

3. `Policy` *responds by a transition* $\langle t_i, e_i^p, t_{i+1}\rangle \in \Delta_{\mathcal{I}}^p$ *such that* $(\mathcal{M}, \alpha_i) \models e_i^p$ *and moves to* t_{i+1}.

The winner of the game is determined by the following rules:

- *If the* `Contract` *cannot move then* `Policy` *wins.*
- *If the* `Policy` *cannot move then* `Contract` *wins.*
- *Otherwise there are two infinite concrete runs* $\overrightarrow{s} = \langle s_0\nu_1 s_1\nu_2 s_2 \ldots\rangle$ *and* $\overrightarrow{t} = \langle t_0\nu_1 t_1\nu_2 t_2 \ldots\rangle$ *respectively of* A^c *and* A^p. *If* $\overrightarrow{s} = \langle s_0\nu_1 s_1\nu_2 s_2 \ldots\rangle$ *is an accepting concrete run for* A^c *and* $\overrightarrow{t} = \langle t_0\nu_1 t_1\nu_2 t_2 \ldots\rangle$ *is not an accepting concrete run for* A^p *then* `Contract` *wins. In other cases,* `Policy` *wins.*

Intuitively in the compliance game, the `Contract` tries to make a concrete move and the `Policy` follows accordingly to show that the `Contract` move is allowed. If the `Policy` cannot move then `Contract` is not compliant, meaning there is a move that the `Policy` can not do, that is that particular action is a violation.

Example 4.1 *In a game between the* `Contract` *from Figure 4a and the* `Policy` *from Figure 4b, the* `Contract` *can choose to invoke the url* `http://www.google.com` *and the* `Policy` *can respond by selecting the appropriate expression which is satisfied by that valuation.*

A more complex situation occurs in the infinite case where infinite runs correspond to liveness properties, i.e. something good will eventually happen. An example of this property is shown in Example 2.3. In this case, the `Contract` only wins (i.e. it breaks the Policy) when according to its view of the world there are infinitely many good things but not for the `Policy` which after some initial good things is trapped in an endless sequence of unsatisfactory states.

Example 4.2 *In a game between the* `Contract` *and* `Policy` *from Ex.2.3, the* `Contract` *can choose to invoke the url* `https://sourceforge.net` *in a certain step after in some previous steps it invokes permission* `io.Connector.https`. *The* `Policy` *can respond by selecting the appropriate expression which is also satisfied by the same valuation, which is possible in the game if* `Policy` *has previously requested permission* `io.Connector.https`.

The *concrete strategy* for `Policy` in game $G^C_{A^c,A^p}(s_0, t_0)$ is a partial function that determines its next move given the history of the concrete game up to a certain point.

Definition 4.2 (Concrete Strategy) *A partial function* $f : S^c \times (S^p \times \nu \times S^c)^* \to S^p$ *is a* concrete strategy *if for any sequence* $\langle s_0\nu_1 s_1\nu_2 \ldots s_i\nu_i s_{i+1}\rangle$ *which is a valid concrete run for* A^c

- $f(s_0) = t_0$
- $f(\langle s_0 t_0 \nu_1 s_1 \dots s_i t_i \nu_{i+1} s_{i+1} \rangle) = t_{i+1}$ *such that* $\langle t_i, e_i^p, t_{i+1} \rangle \in \Delta_{\mathcal{T}}^p$ *and* $(\mathcal{M}, \alpha_i) \models e_i^p$, *where* ν_i *represents* α_i *and* $\mathcal{I}(e_i)$.

A concrete strategy f of a game is a `Policy` *winning strategy* if and only if whenever a `Policy` selects the moves of game as in Definition 4.1 according to f then `Policy` wins.

Definition 4.3 ($\mathcal{AMT}$ Concrete Fair Simulation Relation) *An automaton* A^p concretely fair simulates *an automaton* A^c *if and only if there is a concrete winning strategy for* A^p *we denote as* $A^c \sqsubseteq A^p$. *We also say that* A^c *complies with* A^p.

We have now the machinery to generalize the notion of simulation to symbolic level, among expressions.

Definition 4.4 ($\mathcal{AMT}$ Fair Compliance Game) *A* Fair Compliance Game $G_{A^c, A^p}(s_0, t_0)$ *is played by two players,* `Contract` *and* `Policy`, *in rounds.*

1. *In the first round* `Contract` *is on the initial state* $s_0 \in S^c$ *and* `Policy` *is on the initial state* $t_0 \in S^p$.
2. `Contract` *chooses a transition* $\langle s_i, e_i^c, s_{i+1} \rangle \in \Delta_{\mathcal{T}}^c$ *such that* e_i^c *is satisfiable and moves to* s_{i+1}.
3. `Policy` *responds by a transition* $\Delta_{\mathcal{T}}^p(t_i, e_i^p, t_{i+1})$ *such that* $(e_i^c \rightarrow e_i^p)$ *is valid and moves to* t_{i+1} [1].

The winner of the game is determined by the rules as in Definition 4.1 with the difference in run where we define run over expressions instead of assignments.

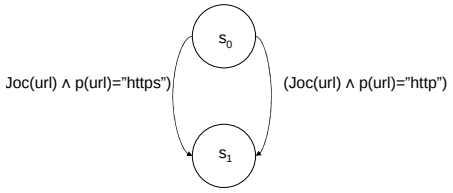
The intuition is similar to concrete game: `Contract` tries to make a symbolic move and the `Policy` follows suit in order to show that the `Contract` move is allowed. If the `Policy` cannot move this means that the `Contract` may not be compliant because there is a symbolic move that the `Policy` could not do. However, as we shall see this might not imply that at the concrete level the `Contract` is really non-compliant.

Definition 4.5 (Strategy) *A partial function* $f : S^c \times (S^p \times E \times S^c)^* \rightarrow S^p$ *is a* symbolic strategy *if and only if for any sequence* $\langle s_0 e_0^c s_1 e_1^c \dots s_i e_i^c s_{i+1} \rangle$ *which is a valid symbolic run for* A^c

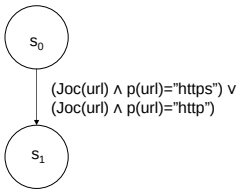
- $f(s_0) = t_0$
- $f(\langle s_0 t_0 e_0^c s_1 t_1 e_1^c \dots s_i t_i e_i^c s_{i+1} \rangle) = t_{i+1}$ *such that* $\Delta_{\mathcal{T}}^p(t_i, e_i^p, t_{i+1})$ *and* $(e_i^c \rightarrow e_i^p)$ *is valid.*

A strategy f of the game is a `Policy` *winning strategy* if and only if whenever a `Policy` select the moves of game as in Definition 4.4 according to f then `Policy` wins.

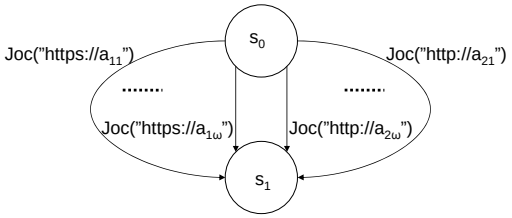
[1] $\rightarrow$ in $(e_i^c \rightarrow e_i^p)$ represents implication symbol in first order logic.



(a) Splitting Edges



(b) Disjuncting Expressions



(c) Concrete Automaton

$$
\begin{aligned}
e_{11} &\doteq (Joc(url) \wedge p(url) = \text{“https”}) \\
e_{12} &\doteq (Joc(url) \wedge p(url) = \text{“http”}) \\
e_2 &\doteq (Joc(url) \wedge p(url) = \text{“https”}) \\
&\quad \vee (Joc(url) \wedge p(url) = \text{“http”})
\end{aligned}
$$

(d) Abbreviations for expressions

Figure 5: Symbolic vs Concrete Automaton

Definition 4.6 ($\mathcal{AMT}$ Fair Simulation Relation) *An automaton A^p* fair simulates *an automaton A^c if and only if there is a winning strategy for A^p we denote as $A^c \leq A^p$. We also say that A^c complies with A^p.*

Proposition 4.1 *If $A^c \leq A^p$ is an $\mathcal{AMT}$ fair simulation relation then $A^c \sqsubseteq A^p$ is a concrete fair simulation relation.*

In contrast to the language inclusion approach where symbolic language inclusion coincides with concrete language inclusion, and also the simulation notions of [18], the converse of Proposition 4.1 does not hold in general.

Proposition 4.2 *$\mathcal{AMT}$ fair simulation is stronger than $\mathcal{AMT}$ language inclusion.*

In order to show that $\mathcal{AMT}$ simulation coincides with concrete simulation we must impose some additional syntactic constraints on the automaton.

Definition 4.7 (Normalized $\mathcal{AMT}$) *$A = \langle E, \mathcal{T}, \Sigma, S, \mathbf{s_0}, \Delta, F \rangle$ is a* normalized automaton *modulo theory $\mathcal{T}$ if and only if for every $s, s_1 \in S$ there is at most one expression $e_1 \in E$ such that $s_1 \in \Delta_{\mathcal{T}}(s, e_1)$.*

For example Figure 5a is a normalized automaton while Figure 5b is not normalized.

Lemma 4.1 *It is possible to normalize an $\mathcal{AMT}$ automaton $A = \langle E, \mathcal{T}, \Sigma, S, \mathbf{s_0}, \Delta, F \rangle$ when theory $\mathcal{T}$ is convex and closed under disjunction.*

Lemma 4.2 *Normalization preserves the determinism of an $\mathcal{AMT}$.*

Proposition 4.3 *For normalized $\mathcal{AMT}$ if $A^c \sqsubseteq A^p$ is a concrete fair simulation relation then $A^c \leq A^p$ is an $\mathcal{AMT}$ fair simulation relation.*

(a) Automaton before normalization (b) Automaton after normalization

Figure 6: Normalization of an automaton

If automata are in normalized form then we have the following theorem from [23]:

Theorem 4.1 *For normalized $\mathcal{AMT}$ $A^c \leq A^p$ is an $\mathcal{AMT}$ fair simulation if and only if $A^c \sqsubseteq A^p$ is a concrete fair simulation.*

5 Simulation Matching

In this section we describe fair simulation for matching and adapts the Jurdziński's algorithm on parity games [21]. The simulation algorithm Algorithm 1 takes as input two automata Aut^C and Aut^P representing respectively the formal specification of a contract and of a policy. A match is obtained when every security-relevant action invoked by Aut^C can also be invoked by Aut^P. In other words, every behavior of Aut^C is also a behavior of Aut^P.

At the first step (line 1) a compliance game graph $G = \langle V_1, V_0, E, l\rangle$ is constructed out of automata Aut^C andAut^P. A compliance game graph can be formally defined as follows:

Definition 5.1 (Compliance Graph) *Given $\langle E^c, \mathcal{T}^c, \Sigma^c, S^c, \mathbf{s_0}^c, \Delta^c_{\mathcal{T}}, F^c\rangle$ and $\langle E^p, \mathcal{T}^p, \Sigma^p, S^p, \mathbf{s_0}^p, \Delta^p_{\mathcal{T}}, F^p\rangle$, construct a $\langle V_1, V_0, E, l\rangle$ as follows:*

- $V_1 = \{v_{(s^c,s^p)} | s^c \in S^c, s^p \in S^p\}$
- $V_0 = \{v_{(s^c,s^p,e^c)} | s^c \in S^c, s^p \in S^p, \exists r^c.s^c \in \Delta^c_{\mathcal{T}}(r^c, e^c)\}$
- $E = \{(v_{(s^c,s^p,e^c)}, v_{(s^c,t^p)}) | t^p \in \Delta^c_{\mathcal{T}}(s^p, e^p) \wedge VALID(e^c \rightarrow e^p)\} \cup$
  $\{(v_{(s^c,s^p)}, v_{(t^c,s^p,e^c)}) | t^c \in \Delta^c_{\mathcal{T}}(s^c, e^c)\}$
- $$l(v) = \begin{cases} 0 & \text{if } v = v_{(s^c,s^p)} \text{ and } s^p \in F^p \\ 1 & \text{if } v = v_{(s^c,s^p)} \text{ and } s^c \in F^c \text{ and } s^p \notin F^p \\ 2 & \text{otherwise} \end{cases}$$

A compliance graph *G is the tuple $\langle V_1, V_0, E, l\rangle$*

Intuitively the compliance level l(v) is 0 when the simulating automaton accepts, 1 when the simulated automaton accepts (but the simulating automaton has not accepted

Algorithm 1 Simulation Algorithm

Input: two $\mathcal{AMT}$ automata Aut^C and Aut^P

Construct compliance game graph $G = \langle V_1, V_0, E, l \rangle$
for all $v \in V$ **do**
  $\mu(v) := \mu_{\texttt{new}}(v) := 0$
end for
repeat
  $\mu := \mu_{\texttt{new}}$
  for all $v \in V_0$ **do**
    $\mu_{\texttt{new}}(v) := \begin{cases} \infty & \text{if} \quad \{\mu(w)|(v,w)\} = \emptyset \\ min\{\mu(w)|(v,w)\} & \text{otherwise} \end{cases}$
  end for
  for all $v \in V_1$ **do**
    $max_v := max\{\mu(w)|(v,w) \in E\}$
    $\mu_{\texttt{new}}(v) := \begin{cases} \infty & \text{if} \quad max_v = \infty \\ 0 & \text{if} \quad l(v) = 0 \\ max_v + 1 & \text{if} \quad l(v) = 1 \\ max_v & \text{if} \quad l(v) = 2 \end{cases}$
  end for
until $\mu = \mu_{\texttt{new}}$
if $\mu(v_{(\mathbf{s_0}^c, \mathbf{s_0}^p)}) < \infty$ **then**
  Simulation exists
end if

yet) and 2 when neither of them accepts. V_1 consists of $v_{(s^c,s^p)}$ where Aut^C is on s^c and Aut^P is on s^p and it is `Contract` turn to move. V_0 consists of $v_{(s^c,s^p,e^c)}$ where Aut^C is on s^c and Aut^P is on s^p, `Contract` just made a move e^c and it is `Policy` turn to move such that $VALID(e^c \rightarrow e^p)$ by querying to an oracle for the SMT solver.

Lemma 5.1 *Let* $Aut^C = \left\langle E^{Aut^C}, \mathcal{T}^{Aut^C}, \Sigma^{Aut^C}, S^{Aut^C}, \mathbf{s_0}^{Aut^C}, \Delta_{\mathcal{T}}^{Aut^C}, F^{Aut^C} \right\rangle$ *and* $Aut^P = \left\langle E^{Aut^P}, \mathcal{T}^{Aut^P}, \Sigma^{Aut^P}, S^{Aut^P}, \mathbf{s_0}^{Aut^P}, \Delta_{\mathcal{T}}^{Aut^P}, F^{Aut^P} \right\rangle$ *be* $\mathcal{AMT}$ *automata and let the theory* $\mathcal{T} = \mathcal{T}^{Aut^C} \cup \mathcal{T}^{Aut^P}$ *be decidable with an oracle for the SMT problem in the complexity class* $\mathcal{C}$

1. $|G = \langle V_1, V_0, E, l \rangle|$ *constructed out of automata* Aut^C *and* Aut^P *by Definition 5.1 is in*
   $O(|S^c| \,.\, |S^p| \,.\, |\Delta_{\mathcal{T}}^c|)^{\mathcal{C}}$
2. $\left|l^{-1}(1)\right|$ *defined as in Definition 5.1 is in* $O(|S^c| \,.\, |S^p|)$

A compliance game $P(G, v_0)$ on G starting at $v_0 \in V$ is played by two players `Policy` (for Aut^P) and `Contract` (for Aut^C). The game starts by placing pebble on v_0. At round i with pebble on v_i, $v_i \in V_0(V_1)$, `Policy` (`Contract` resp.) plays and moves the pebble to v_{i+1} such that $(v_i, v_{i+1}) \in E$. The player who cannot move loses. For infinite play $\pi = v_0 v_1 v_2 \ldots$, the winner defined as the minimum compliance level that

occurs infinitely often, namely if the minimum compliance level is 0 or 2 then `Policy` wins, otherwise `Contract` wins.

Next, we define a *compliance measure* $\mu : V \rightarrow \{x | x \leq |l^{-1}(1)|\} \cup \{\infty\}$. μ ranges from 0 to $|l^{-1}(1)|$ because at l(v)=1 the simulated automaton (contract) accepts but the simulating automaton (policy) has not accepted yet. Thus, progressing the measure has the analogy of computing the pre-fixed point where the `Contract` remains winning and ∞ shows that the μ keeps progressing beyond this limit, meaning `Contract` wins the game. If $l(v) = 1$, then $\mu(v) > \mu(w)$, where $|l^{-1}(1)| + 1 = \infty$. If $l(v) = 2$ or $l(v) = 0$, then $\mu(v) \geq \mu(w)$.

The compliance measure for each node is the number of potential bad nodes, namely nodes where the contract accepts but the policy does not, that it can reach. Thus, $\mu(v) = \infty$ means that there is an infinite path where policy cannot return to compliance level 0. We slighty modify the Jurdziński progress measure [21] to *compliance measure* where instead of a pair $(0, x)$ we only use x. This is due to our observation of our domain where we only have three priorities, namely $l(v) \in 0, 1, 2$ thus for ordering $(0, x) \geq_{l(v)} (0, x')$ the first component will not effect the ordering.

Jurdziński's algorithm on parity games [21] defines that `Policy` has a winning strategy from precisely the vertices v where after its lifting algorithm halts has $\mu(v) < \infty$. However, in contract-policy matching we are interested when there is a winning strategy from the initial vertex $v_{(\mathbf{s_0}^c, \mathbf{s_0}^p)}$, depicted in Algorithm 1 as $\mu(v_{(\mathbf{s_0}^c, \mathbf{s_0}^p)}) < \infty$.

Proposition 5.1 *Let G be a parity game constructed from two $\mathcal{AMT}$ automata Aut^C and Aut^P constructed as in Definition 5.1.* `Policy` *has a winning strategy from the initial vertex $v_{(\mathbf{s_0}^c, \mathbf{s_0}^p)}$ when Algorithm 1 halts with $\mu(v_{(\mathbf{s_0}^c, \mathbf{s_0}^p)}) < \infty$.*

6 The Architecture

In this section we describe the conceptual architecture of the prototype that implements the overall matching algorithm and supports integration with state of the art decision procedure solver NuSMV [9] integrated with its MathSAT libraries [7]. The main aim is to provide a concrete overview of how the prototype is implemented so that one can easily understand the possible options for integration with the solver. The contract-matching prototype takes as input a contract and a policy both specified in ConSpec and checks whether or not the contract matches the policy. The source code itself is thoroughly documented and should therefore be easy to understand. In addition, the following class diagram should provide the reader with a good overview over the Simulation Algorithm namespace and its classes as shown in Figure 7. Detailed class diagram is available on Appendix A.

The prototype had been implemented as a Desktop version by extending the prototype from [6]. The prototype consists of only one part which is off-device implementations. At the first step of matching, a compliance game graph $G = \langle V_1, V_0, E, l \rangle$ is constructed out of automata Aut^C and Aut^P. The main parity game algorithm runs on the constructed game graph and makes calls to the decision procedure during its execution. The different step from the on-the-fly implementation is that the policy automata need not be complemented. The rest of integration issues with decision solver based on MathSAT and NuSMV follows from on-the-fly matching implementation, for example

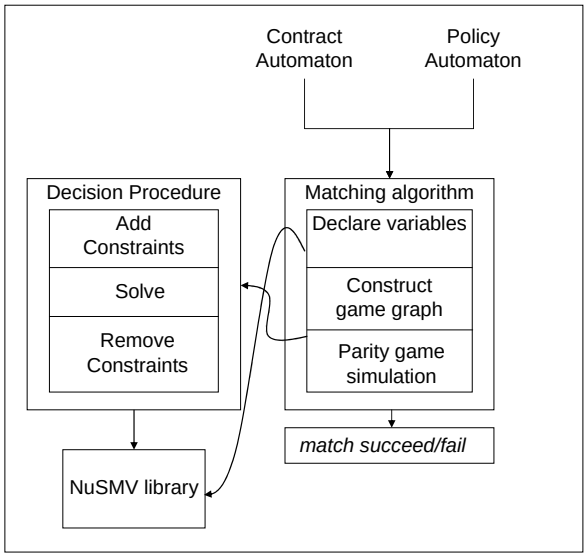


Figure 7: Simulation Implementation Architecture

we use the solver as a black box (an oracle) for the general algorithm that gives the answer whether the problem is satisfiable or not.

The prototype is basically separated in 2 parts: on-device and off-device implementations. During off-device part execution, the contract and policy are transformed into a suitable internal representation for the on-the-fly algorithm. In on-device part of the prototype the main on-the-fly algorithm runs over already created contract and policy as AMT and makes a significant amount of calls to the decision procedure during execution.

The initial parsing algorithm just transforms a contract (resp. a policy) into a .NET class, ContractAutomaton.cs (resp. PolicyAutomaton.cs) that can be directly manipulated by the actual algorithm responsible for the simulation matching.

Since a contract-policy matching algorithm should frequently call the decision procedure during its runs, we have found a design decision for an internal representation of $\mathcal{AMT}$. This particular form of $\mathcal{AMT}$ supports all the options of integration with solver that we address in this paper.

A number of variables is associated to every edge, where *method* is an API call that the policy is supposed to rule, *cond* - a guarded command which must be true in order for the method to be executed, for instance a *cond* specifies that the url must start with the string "https".

For further representation simplification, we follow the semantics for security automata proposed in [1] so that we have a prioritized execution among guards: we go to the next guard only if the guards before it have all failed. Such information is represented in *otherConds* - the other guarded commands that failed before reaching the current guard $otherMethods$ - an expression consists of all other methods that are not supposed to rule at the current moment e.g. $\neg m_1 \wedge \neg m_2$ where m_1 and m_2 are methods that are not supposed to rule.

We use the solver as a black box for the general algorithm so it gives the answer whether the problem is satisfiable or not. In this way it could be easy to also try a different decision procedure such as MathSAT by Bozzano et al. [7], DPLL(T) by Tinelli

[25] or CVC-lite [2]. For the same reason we have further decided to interface with the solver without using its internal data structure but rather to interact with the decision procedure by using strings. While this creates a bit of overhead for parsing, it makes it significantly easier to replace the solver. An industry level application committing to a particular solver would likely bypass this step.

Among the different possibilities we have used the decision procedure libraries behind the tools MathSAT and NuSMV [9]. In this way we could support expressions in the edges of the automaton modulo theory that are arbitrarily complex boolean expression, mathematical expression and uninterpreted function symbols.

Remark 6.1 *All the paths given in this technical report are from internal repository, for publicly available prototype the path may differ and will be explained in the corresponding distribution.*

The SMA was first implemented as a Desktop version by extending the existing S3MSDesktopMatcher project which can currently be found in the following directory: s3ms_code\Automata_Matching\Release_1.10\NET_DESKTOP_Matcher.

In the following the most important classes of the implementation will be described:

Simulation.cs This is the main class of the SMA implementation. It contains the Run() method that can be used to execute the algorithm.

SimulationDebugHelper.cs A helper class that, if used, prints a lot of useful debug information about the execution of the algorithm.

SimulationTester.cs This class is used for testing the algorithm within the Desktop environment. It contains various test cases that can be enabled. Notice that the test cases need to be uncommented directly in the source code, since there can only be on instance of the NUSMV solver and therefore only one execution of a test case per run of the program. This class is only meant for initial testing and testing of special test cases, since only one test case can be executed at a time. An external testing framework was created to run multiple tests at once.

6.1 Updating the S3MS ConSpec Parser

The S3MS ConSpec Parser is updated from the previous implementation in [6] such that it supports complementing the policy automata only if SMA should not be used, instead of complementing the policy automaton in every case. A command line argument was added to the Parser that specifies whether the policy automaton should be complemented. The following files were updated:

- The file Program.cs was changed to check if the third command line argument is set to "false". In this case the automaton is not complemented.
- The file \Business\CSContent.cs was updated to only complement the automaton if the command line argument was not set or set to true.

The new version of the parser can be found in the directory:
s3ms_code\ConSpec_Parser\Simulation_Algorithm_Update\Conspec\FOR_NET_DESKTOP\S3MS-Parser\S3MS-VBProject.

Table 2: Benchmark Contract and Policies

Example ID	Natural Language description	Coverage
httpHttps	The application only uses high-level network connections.	NET
https	The application only uses HTTPS network connections.	NET, PRI
maxKB512	The data received by application is bounded by 512Kb	USE, NET
maxKB1024	The data received by application is bounded by 1024Kb	USE, NET
noPushRegistry	The application does not use the push registry mechanism	USE
oneConnPushRegistry	Only one connection registered to the Push registry at a time	USE, NET
notCreateRSt	The policy allows to open record stores, but it is not allowed to create new record stores.	INT
notCreateSharedRS	The application does not create shared record stores.	INT, PRI
noSMS	No messages are sent by the application	USE
100SMS	Maximum 100 text messages can be sent by the application	USE
pimNoConn	After PIM was opened no connections are allowed	USE, PRI, NET
pimSecConn	After PIM was accessed only secure connections (HTTPS) can be opened	USE, PRI, NET

6.2 Extending the S3MS PolicyManager

The GUI of the S3MS PolicyManager has been updated to be able to create a policy that can be used for Simulation Matching. To create a policy for Simulation Matching the user simply selects the entry "SimulationMatching Representation" in the policy management window. A new version of the parser was plugged-into the S3MS PolicyManager and the S3MS PolicyManager was updated to make use of its new functionality. The actual version of the S3MS PolicyManager can be found in:
s3ms_code\Simulation_Algorithm_Framework_Update\S3MS.PolicyManager

6.3 Extending the S3MS Framework

The S3MS Framework itself had to be extended to support the SMA. This work is currently in progress. At this point the S3MS Framework has already been prepared to support the SMA, but the old version of the S3MS Matcher has to be replaced with the new version of the S3MS Matcher that implements SMA, before SMA can be used within the framework. However, the current SMA implementation was not yet plugged-into the framework. To plug SMA implementation into the framework, first the existing S3MS DesktopMatcher project must be compiled for the use in a mobile device. Next, the old Matcher executable in the Framework must be replaced with the new version. The old version of the S3MS Matcher that has to be replaced can be found in:
s3ms_code\Simulation_Algorithm_Framework_Update\S3MS.PolicyManager\S3MS.SemanticMatcher\SEMANTIC_BIN

6.4 Test Cases

Test cases based on matching algorithm with language inclusion implementation [6]. We consider a number of examples for experiments that provide a good coverage of the requirements that we mentioned afore (Table2). We append to each problem name the _contract or _policy suffix denoting whether the rule is used to specify a contract or a policy.

The ConSpec files, CS files, DLL files and an overview over all the test cases can be found in:

s3ms_code\S3MS_Testing\Simulation_Algorithm\TestCases.

7 Design Decisions

In integrating matching algorithm with the theory solver we faced a number of design options, where different design decisions are made in order to *decide the best configuration of integrating automata-based inclusion algorithm with decision procedure* as the problem is not trivial. Every option of the configuration proposed below has different memory impact and this information and results of such analysis is very important because of the resource constraints of mobile device. In integrating matching algorithm with the theory solver we faced a number of design options:

One_vs_Many Solver in object oriented languages is by itself an object. We could either create only one instance of solver, relying on the solver to assert and retract expressions on demand, or create a new instance of the solver every time we call the decision procedure.

ALL_INSTANCES The expression sent to the solver has the following structure: $method \wedge otherMethods \wedge cond \wedge otherConds$.

CACHING_MC Since many edges will be traversed again and again we could save time by caching the results of the matching. The solver itself has a caching mechanism that could be equally used (CACHING_SOLVER).

Unlike in on-the-fly matching implementation, we do not have MUTEX_SOLVER, MUTEX_MC, and PRIORITY_MC options instead we introduce ALL_INSTANCES which is suitable for representation of only policy automaton and not the complementation of policy automaton.

As in on-the-fly matching implementation, the One_vs_Many option was not possible which requires only one instance of solver exists at time in order to interact correctly with the NuSMV library. This leads to use a static invocation for the solver and set significant constraints on the interaction. For example, before starting to visit all constraints to the library, all variables used in expressions must be declared. The NuSMV library has to invoke *DeclareNewBooleanVar, DeclareNewWordVar, DeclareNewStringVar* methods for declaration of boolean, integer and string variables respectively. Only after declaring all the variables from contract and policy expressions, the simulation algorithm can actually start invoking the decision procedure in its visit. A consequence of this rule is that with this implementation we cannot insert edges that introduce new variables because the solver can be called only after declaring all the variables and adding all the needed constraints. Therefore, during the visit of the algorithm we must at first upload constraints to the solver with the *AddConstraint* method of the NuSMV class and then remove them with the *RemoveConstraint*.

Therefore, during the visit of the algorithm we must at first upload constraints to the solver with the *AddConstraint* method of the NuSMV class and then remove them with the *RemoveConstraint*.

Table 3: Problems Suit

Problem	Contract	Policy	SC	TC	SP	TP
P1	size_100_512_contract.pol	size_10_1024_policy.pol	2	4	2	4
P2	maxKB512_contract.pol	maxKB1024_policy.pol	2	4	2	4
P3	noPushRegistry_contract.pol	oneConnRegistry_policy.pol	2	3	3	9
P4	notCreateRS_contract.pol	notCreateSharedRS_policy.pol	2	4	2	4
P5	pimNoConn_contract.pol	pimSecConn_policy.pol	3	7	3	9
P6	2hard_contract.pol	2hard_policy.pol	3	7	3	7
P7	http_contract.pol	https_policy.pol	3	7	3	7
P8	3hard_contract.pol	3hard_policy.pol	3	7	3	7
P100	noSMS_contract.pol	100SMS_policy.pol	2	4	102	304

SC: Number of States Contract TC: Number of Transitions Contract
SP: Number of States Policy TP: Number of Transitions Policy

(a) Abbreviations

Table 4: Running Problem Suit 10 Times

ALL_INSTANCES ONE_INSTANCE CACHING_MC			
Problem	**ART (s)**	**CRT (s)**	**Result**
P1	2.014	2.014	Match
P2	1.934	3.948	Match
P3	1.886	5.834	Match
P4	1.886	7.72	Match
P6	1.998	1.998	Not Match
P7	2.06	4.058	Not Match
P8	1.998	6.056	Not Match
P100	5.528	5.528	Match

(a) Running Problem Suit

ART: Average Runtime for 10 runs
CRT: Cumulative Average Runtime

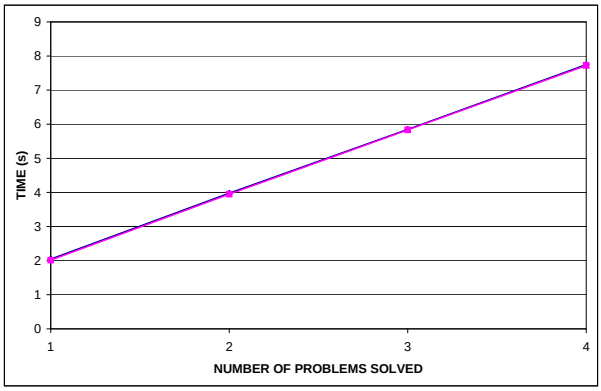
(b) Abbreviations

8 Experiments on Desktop

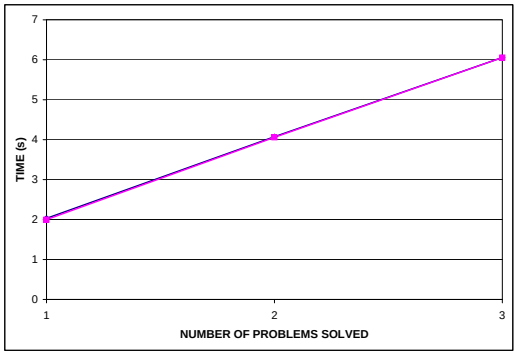
To understand the best option we collected data on running time for each problem in each design alternative and the number of solved problems against time. From (Section 7) the design alternatives can be implemented and tested in two alternative configurations and we use the same problem suit as in Table3 for possible combinations of policy-contract (mis)matching pairs.

We run our experiments on a Desktop PC (Intel(R) Pentium(R) D CPU 3.40GHz, 3389.442MHz, 1.99GB of RAM, 2048 KB cache size) with operating system Microsoft Windows XP Professional Version 2002 Service Pack 3. The result is shown in Table 4.

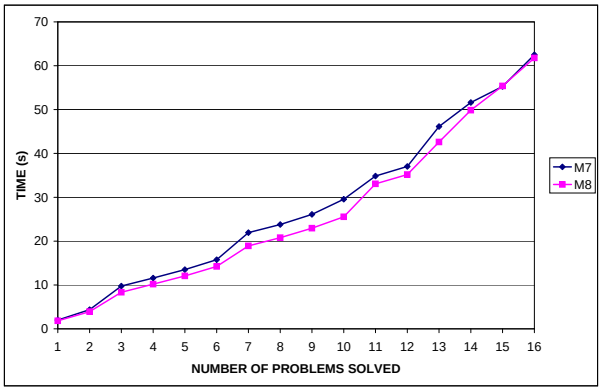
For the sake of example we present the result obtained for alternative with ALL_INSTANCES ONE_INSTANCE CACHING_MC in Table 4. The results for all design alternatives are mapped into diagram shown in Figure 8a for matching problems and Figure 8b for not matching problems. Notice that we only provide the cumulative running time that is necessary to solve all problems as for on-the-fly implementation experiments. This is important because our goal is to match (or not match) all rules in a contract with all corresponding



(a) Match succeeds for real policies



(b) Match fails for real policies



(c) Matches among synthetic contracts and policies

M7: ALL_INSTANCES ONE_INSTANCE CACHING_SOLVER
M8: ALL_INSTANCES ONE_INSTANCE CACHING_MC

(d) Abbreviations for Configurations

Figure 8: Cumulative response time of matching algorithm on Desktop PC

rules in a policy. Thus, the value of the single problem is not important except for some cases where the average output might be significantly off due to some off scale rule.

We singled out P100 as a challenging artificial problem because it has a large number of states compared to the others: essentially this happened because we draw an automaton modulo theory with 100 states and which traverse from one state to another by adding 1 to the number of SMS sent.

In this case there is a difference between M7 and M8, namely 5.387 s and 4.434 s resp., that is M8 is better around 21.5% than M7. In order to study this anomaly in more details, we generated more unreal problem sets: as P100 with combination of sent SMS none, 1, 10, and 100 for both contract and policy. The data of the experiment is given on Appendix B. The generated cases cumulative running time of implementation is propositional to the number of problems solved (see Figure 8c). In this case the difference between M7 and M8 is only around 9.8% still with M8 better than M7. This result conforms to our intuition because M8 uses fewer calls to solver due to its caching and thus save computations.

All methods seem to perform equally well because the problems are not stressful

enough for the different configurations. This is actually a promising result for the deployment to the resource constrained in mobile device domain. However, we have not yet implemented the same algorithm for a mobile platform.

In this chapter, we have given possible design decisions and run experiment on PC for $\mathcal{AMT}$ simulation. Furthermore, we have detailed the time of the running on the mobile platform for one design decision only to give the reader a feeling how the matching algorithm with integrated decision procedure can run in real life and that it will take a reasonable time. Our current implementation uses ALL_INSTANCES ONE_INSTANCE CACHING_MC configuration. ALL_INSTANCES is preferred because of the nature of rules in policies when an automaton is not complemented. ONE_INSTANCE is chosen because of garbage collection problem. CACHING_MC is desired in order to save calls to solver for the already solved rules.

As already described on (Section7) the S3MSDesktopMatcher project can only run one test case at a time, since there can only be one instance of the NuSMV solver. Therefore an external TestSuite was created to run all the test cases at once. This Testsuite is located in:

s3ms_code\S3MS_Testing\Simulation_Algorithm\SimulationTestSuite

In the following the most important classes of the TestSuite will be described:

SimulationTester.cs This is the main class of the TestSuite. Use the Run() method to execute the test cases. The file also contains some of the settings that have to been customized according to your system. Most importantly the variable MatcherLocation has to be modified to point to the executable of the S3MSDesktopMatcher project.

SimulationTestCase.cs A test case that is going to be executed. In this class the flags can be adjusted that will be used to execute the S3MSDesktopMatcher.

Program.cs This is the class that will be executed when the TestSuite is run. It contains information about all the command line arguments that are accepted by the TestSuite.

We run our experiments on a Desktop PC (Intel(R) Pentium(R) D CPU 3.40GHz, 3389.442MHz, 1.99GB of RAM, 2048 KB cache size) with operating system Microsoft Windows XP Professional Version 2002 Service Pack 3. The result is shown in Table 5.

9 Related Works and Conclusions

Mobile code security can be achieved by several approaches, for example *code signing* to ensure the origin of the code by trust relationship, *proof-carrying code (PCC)* to ensure safety by explicit proof, *model-carrying code (MCC)* that carries security-relevant behavior of the producer mobile code [27], and *security-by-contract (SxC)* where a digital signature should not just certify the origin of the code but rather bind together the code with a contract [11].

Security-by-contract (SxC). Security-by-contract [11] proposed to augment mobile code with a claim on its security behavior that can be matched against a mobile platform policy on-the-fly, which provides semantics for digital signatures on mobile code.

Table 5: Running Problem Suit 10 Times

ALL_INSTANCES ONE_INSTANCE CACHING_MC			
Problem	**ART (s)**	**CRT (s)**	**Result**
P1	2.014	2.014	Match
P2	1.934	3.948	Match
P3	1.886	5.834	Match
P4	1.886	7.72	Match
P6	1.998	1.998	Not Match
P7	2.06	4.058	Not Match
P8	1.998	6.056	Not Match
P100	4.53	4.53	Match

(a) Running Problem Suit
ART: Average Runtime for 10 runs CRT: Cumulative Average Runtime

(b) Abbreviations

Security-by-contract attempts to overcome the major limitation of MCC, namely not fully developed issue of contract matching and limited to finite state automata which are too simple to describe realistic policies. In coping with this challenge, we propose an application of formal methods that goes beyond the traditional realm of off-line verification of formal properties of hardware and software. The formal model considered for capturing contracts and policies is based on the novel concept of *Automata Modulo Theory* ($\mathcal{AMT}$).

Off-line Verification. Our approach is different from off-line verification while we use integration of emptiness test for automata modulo theory with satisfiability using decision procedures. Such reasoning capabilities should then be used at the time an application is downloaded on a mobile application such as PDA or a smart phone. The usage of decision procedures allowed us to cope with automata modulo theories where edges are not just finite states of labels but rather expressions that can capture infinite transitions such as "connect only to urls starting with https://". In the off-line verification realm, the idea of embedding decision procedures into a higher level reasoner is well accepted and was one of the strongholds of the PVS system. At theoretical level Tinelli in [30] combines order-sorted first-order theories and their decision procedures for theories satisfying certain conditions into a decision procedure for their union, where SMT problems themselves can be addressed by tools such as CVC [2], UCLID [8], MathSAT [7].

Infinite States System. Infinite numbers of transitions in security policies by labeling each transition with a computable predicate instead of an atomic symbol has been studied in [26] and implemented in systems like PoET/PSLang toolkit [13]. Edit automata [3] extend security automata to model the transforming effects of in-lined reference monitors and is implemented in the Polymer system [4]. These approaches focus on the relations between code and security claims on the code. The Mobile system [17] implements a linear decision algorithm that verifies that annotated .NET bytecode binaries satisfy a class of policies that includes security automata and edit automata.

Conclusions. We have described the prototype implementation, its integration with a state of the art decision solver (based on MathSAT and NuSMV) and the preliminary experiments that we have done for contract-policy matching.

Acknowledgments

We thank S. Vogl for first version of simulation prototype implementation. We also acknowledge N. Bielova for comments and suggestions regarding interaction with solver and for support in the implementation.
The EU-FP6-IST-STREP-S3MS project for partly supporting this research.

A Simulation Matching Prototype Class Diagram

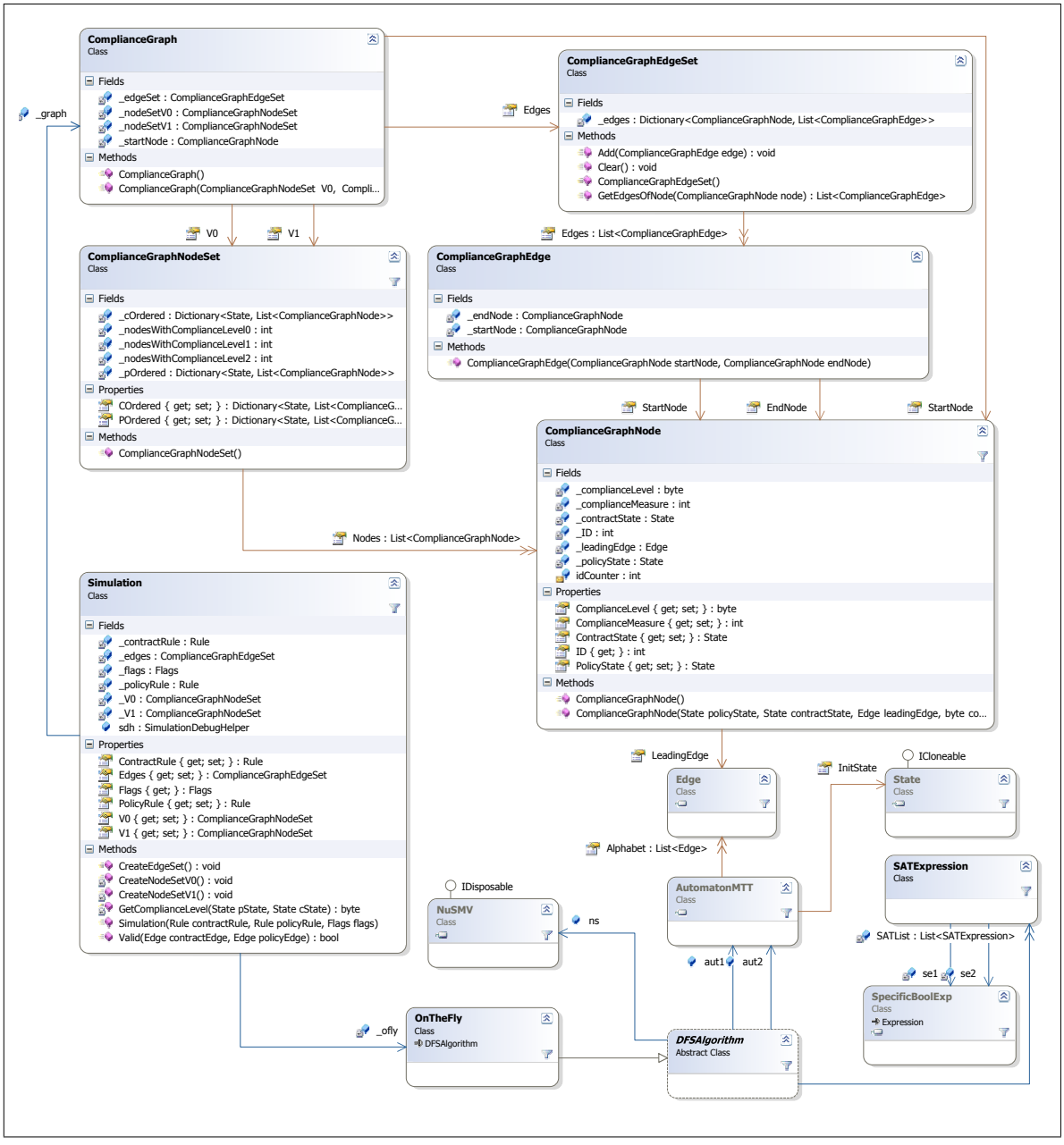


Figure 9: Simulation Class Diagram

B Simulation Matching Prototype Experiments

Table 6: Problems Suit

Problem	**Contract**	**Policy**
P100-100	100SMS_contract.pol	100SMS_policy.pol
P100-10	100SMS_contract.pol	100SMS_policy.pol
P100-1	100SMS_contract.pol	100SMS_policy.pol
P100-NO	100SMS_contract.pol	noSMS_policy.pol
P10-100	10SMS_contract.pol	100SMS_policy.pol
P10-10	10SMS_contract.pol	10SMS_policy.pol
P10-1	10SMS_contract.pol	1SMS_policy.pol
P10-NO	10SMS_contract.pol	noSMS_policy.pol
P1-100	1SMS_contract.pol	100SMS_policy.pol
P1-10	1SMS_contract.pol	10SMS_policy.pol
P1-1	1SMS_contract.pol	1SMS_policy.pol
P1-NO	1SMS_contract.pol	noSMS_policy.pol
PNO-100	noSMS_contract.pol	100SMS_policy.pol
PNO-10	noSMS_contract.pol	10SMS_policy.pol
PNO-1	noSMS_contract.pol	1SMS_policy.pol
PNO-NO	noSMS_contract.pol	noSMS_policy.pol

Table 7: Average Running Problem Suit 10 Times (s)

Problem	**M7**	**M8**	**Result**
P100-100	3.668	5.528	Match
P100-10	5.465	7.259	Not Match
P100-1	9.106	7.419	Not Match
P100-NO	7.228	6.385	Not Match
P10-100	5.308	7.531	Match
P10-10	3.446	2.59	Match
P10-1	2.308	2.165	Not Match
P10-NO	2.18	2.105	Not Match
P1-100	6.184	4.696	Match
P1-10	2.26	2.15	Match
P1-1	1.918	1.886	Match
P1-NO	1.854	1.87	Not Match
PNO-100	5.387	4.434	Match
PNO-10	2.372	2.077	Match
PNO-1	1.995	1.838	Match
PNO-NO	1.822	1.838	Match

References

[1] I. Aktug and K. Naliuka. Conspec - a formal language for policy specification. In *Proc. of the 1st Int. Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)*, Dresden, Germany, 2007.

[2] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. of the 16th Int. Conf. on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 515–518. Springer-Verlag, 2004.

[3] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Found. of Comp. Security*, 2002.

[4] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *Proc. of the ACM SIGPLAN 2005 Conf. on Prog. Lang. Design and Implementation*, pages 305–314. ACM Press, 2005.

[5] N. Bielova, N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Matching in security-by-contract for mobile code. *J. of Logic and Algebraic Programming*, 78:340–358, May-June 2009.

[6] N. Bielova, F. Massacci, and I. Siahaan. Testing decision procedures for security-by-contract. In *Joint Workshop on Found. of Comp. Sec., Automated Reasoning for Sec. Protocol Analysis and Issues in the Theory of Sec. (FCS-ARSPA-WITS'08)*, 2008.

[7] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P.v. Rossum, and R. Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. *J. of Autom. Reas.*, 35(1):265–293, 2005.

[8] R. E. Bryant, S.K. Lahiri, , and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV'02)*, LNCS, pages 78–92. Springer-Verlag, 2002.

[9] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV'02)*, LNCS, pages 359–364. Springer-Verlag, 2002.

[10] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .NET platform. *Information Security Tech. Rep.*, 13(1):25 – 32, 2008.

[11] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code. In *Proc. of the 4th European PKI Workshop Theory and Practice (EUROPKI'07)*, page 297. Springer-Verlag, 2007.

[12] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Department of Computer Science, Cornell University, 2004.

[13] U. Erlingsson and F.B. Schneider. IRM enforcement of Java stack inspection. In *Proc. of the 2000 IEEE Symp. on Security and Privacy*, pages 246–255, 2000.

[14] K. Etessami, T. Wilke, and R. Schuller. Fair simulation relations, parity games, and state space reduction for büchi automata. *SIAM J. on Comp.*, 34(5):1159–1175, 2005.

[15] S. Gurumurthy, R. Bloem, and F. Somenzi. Fair simulation minimization. In *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV'02)*, pages 610–624. Springer-Verlag, 2002.

[16] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[17] K.W. Hamlen, G. Morrisett, and F.B. Schneider. Certified in-lined reference monitoring on .net. In *Proc. of the 2006 workshop on Prog. Lang. and analysis for security*, pages 7–16. ACM Press, 2006.

[18] M. Hennessy and H. Lin. Symbolic bisimulations. In *MFPS'92: Selected papers of the meeting on Math. Foundations of Programming Semantics*, pages 353–389. Elsevier Sci. Publishers B. V., 1995.

[19] T.A. Henzinger, O. Kupferman, and S.K. Rajamani. Fair simulation. In *Proc. of of the 8th Int. Conf. on Concurrency Theory*, pages 273–287. ACM Press, 1997.

[20] M. Hilty, A. Pretschner, C. Schaefer, and T. Walter. Usage control requirements in mobile and ubiquitous computing applications. In *Proc. of the Int. Conf. on Sys. and Net. Comm. (ICSNC 2006)*, pages 27–27. IEEE Press, 2006.

[21] M. Jurdzinski. Small progress measures for solving parity games. In *STACS '00: Proc. of the 17th Annual ACM Symposium on Theoretical Aspects of Computer Science*, pages 290–301. Springer-Verlag, 2000.

[22] F. Massacci and I. Siahaan. Matching midlet's security claims with a platform security policy using automata modulo theory. In *Proc. of the 12th Nordic Workshop on Secure IT Systems (NordSec'07)*, 2007.

[23] F. Massacci and I. Siahaan. Simulating midlet's security claims with automata modulo theory. In *Proc. of the 2008 workshop on Prog. Lang. and analysis for security*, pages 1–9, 2008.

[24] G.C. Necula. Proof-carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang.*, pages 106–119. ACM Press, 1997.

[25] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *J. of the ACM*, 53(6):937–977, 2006.

[26] F.B. Schneider. Enforceable security policies. *ACM Trans. on Inf. and Syst. Security*, 3(1):30–50, 2000.

[27] R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D.C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proc. of the 19th ACM Symp. on Operating Syst. Princ.*, pages 15–28. ACM Press, 2003.

[28] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Inform. and Comp.*, 206(2-4):158–184, 2007.

[29] MOBIUS Project Team. Framework- and application-specific security requirements. Public Deliverable D1.2, Mobility, Ubiquity and Security - MOBIUS, 2006. Report available at http://mobius.inria.fr.

[30] C. Tinelli and C.G. Zarba. Combining decision procedures for sorted theories. *LNCS*, pages 641–653, 2004.

[31] S. Vogl. Simulation algorithm project report. Technical report, University of Trento, 2009.

[32] B.S. Yee. A sanctuary for mobile agents. In J. Vitek and C.D. Jensen, editors, *Secure Internet Programming*, pages 261–273. Springer-Verlag, 1999.

[33] A. Zobel, C. Simoni, D. Piazza, X. Nunez, and Daniel Rodriguez. Business case and security requirements. Public Deliverable D5.1.1, EU Project S3MS, 2006. Report available at www.s3ms.org.